DOCUMENTAȚIE

TEMA NUMĂRUL_2

GESTIUNEA COZILOR UTILIZÂND THREAD-URI

NUME STUDENT: ARDELEAN RALUCA-NICOLETA

GRUPA: 30227

Cuprins

1. Obiectivul temei

• Obiectivul principal

Obiectivul principal al acestei teme este proiectarea și implementarea unei aplicații care are ca scop analiza sistemelor bazate pe cozi de așteptare, simulând o serie de N clienți care sosesc pentru servire, intră în cozile de așteptare, așteaptă, sunt serviți și în cele din urmă părăsesc cozile de așteptare. De asemenea, această analiză presupune și calcularea timpului mediu de așteptare, a timpului mediu de procesare și a orelor de vârf, timp în care cozile sunt cele mai aglomerate.

• Obiectivele secundare

Pentru îndeplinirea obiectivului principal au fost urmați următorii pași:

- Analiza problemei și identificarea cerințelor/cazuri de utilizare.
- Proiectarea simulării unor cozi de așteptare.
- Implementarea propriu-zisa a aplicației care execută simularea.
- Testarea aplicației rezultate.

Aceste aspecte vor fi detaliate în capitolele următoare.

2. Analiza problemei, cazuri de utilizare

• Analiza problemei

Cozile sunt utilizate în mod obișnuit pentru a modela domeniile lumii reale. Obiectivul principal al unei cozi este de a oferi un loc pentru ca un „client” să aștepte înainte de a primi un „serviciu”. Managementul pe bază de coadă-sisteme sunt interesate să minimizeze timpul pe care „clienții” îl așteaptă în cozi înainte de servire. O modalitate de a minimiza timpul de așteptare este existența mai multor servere, adică mai multe cozi în sistem (fiecare coadă este considerată ca având un procesor asociat), dar această abordare crește costurile furnizorului de servicii.

• Cerințe funcționale

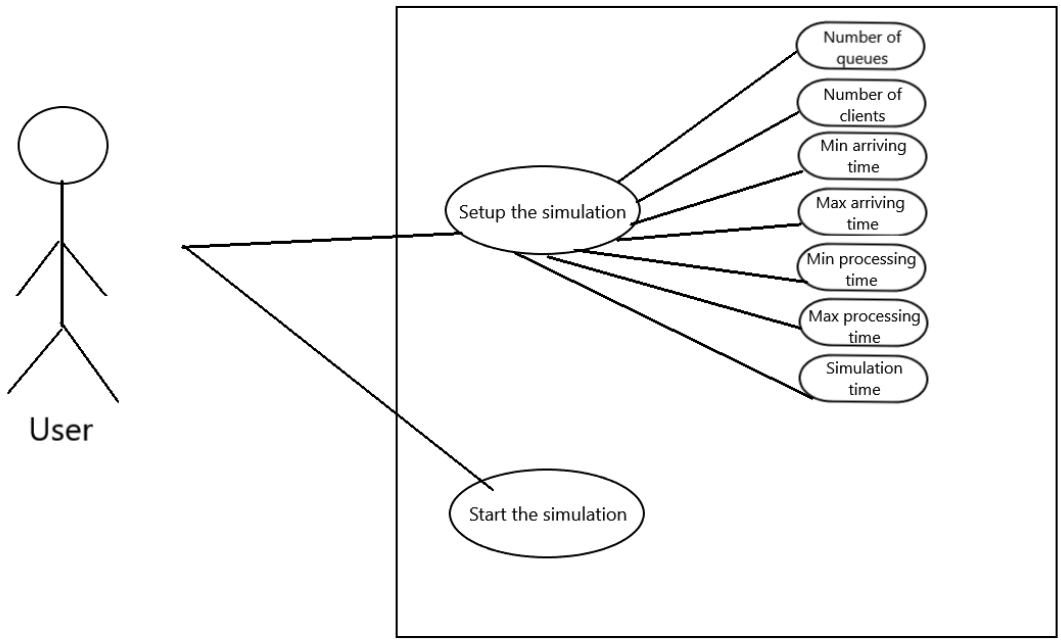
- Aplicația de simulare trebuie să permită utilizatorului să stabilească timpul de simulare.
- Aplicația de simulare trebuie să permită utilizatorului să stabilească numărul de cozi .
- Aplicația de simulare trebuie să permită utilizatorului să stabilească numărul de clienți care au venit pentru servire și care se vor intra în cozile de așteptare.
- Aplicația de simulare trebuie să permită utilizatorului să stabilească un interval pentru timpul de sosire prin setarea valorii minime și maxime.
- Aplicația de simulare trebuie să permită utilizatorului să stabilească un interval pentru timpul de procesare prin setarea valorii minime și maxime.
- Aplicația de simulare trebuie să permită utilizatorului să pornească simularea.

- Aplicația de simulare trebuie să afișeze evoluția în timp a cozilor.

• Cerințe nonfuncționale

- Aplicația de simulare trebuie să fie intuitivă și ușor de utilizat.

• Cazuri de utilizare



1. Utilizatorul introduce valori pentru numărul de cozi, numărul de clienți, intervalul timpului de sosire(timpul minim și timpul maxim), intervalul timpului de procesare(timpul minim si timpul maxim) și timpul de simulare.

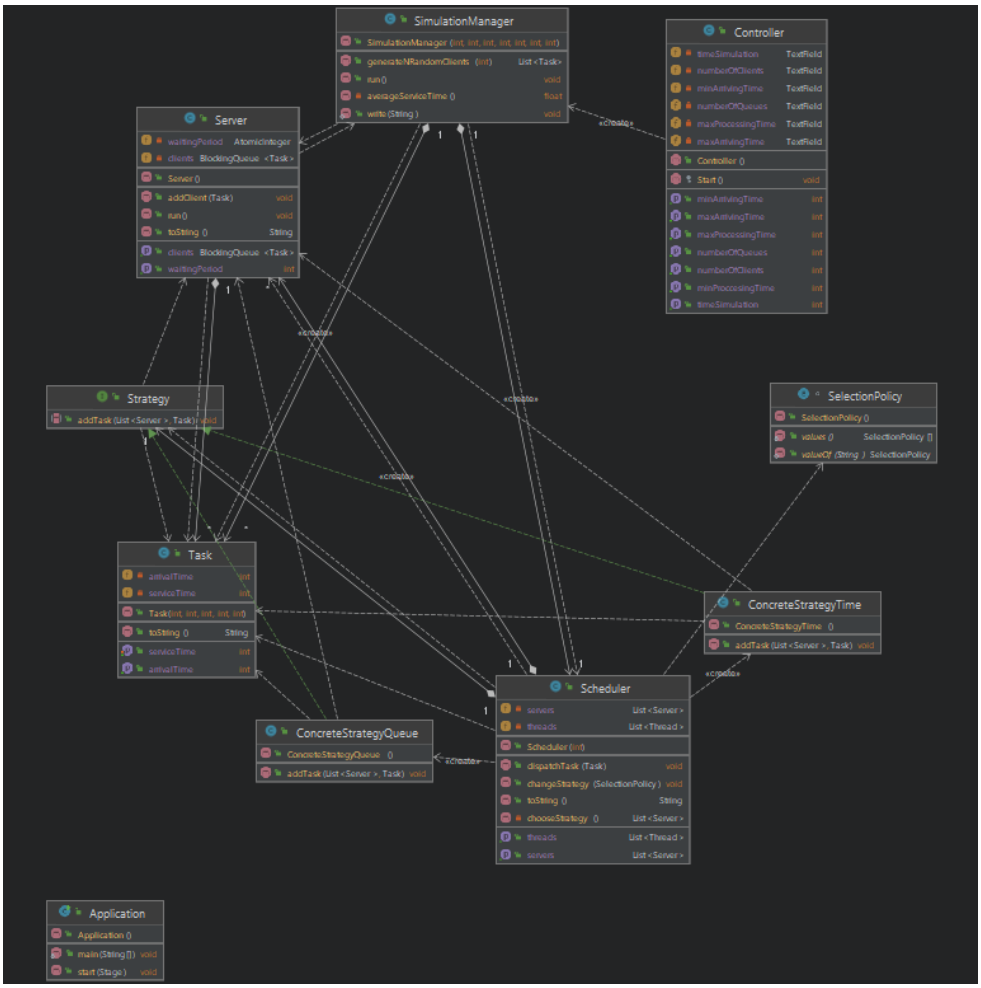
2. Utilizatorul apasă pe “START” și simularea începe dacă datele introduse sunt valide, afișând în timp real starea cozilor la fiecare timp.

Scenariu alternativ: Valori invalide pentru parametrii simulării

Dacă utilizatorul introduce valori pentru capetele intervalelor în ordine inversă, adică dacă timpul minim de sosire este mai mare decât timpul maxim de sosire sau dacă timpul minim de servire este mai mare decât timpul maxim, aplicația va afișa un mesaj de eroare, iar utilizatorul va trebui să reintroducă datele.

3. Proiectarea

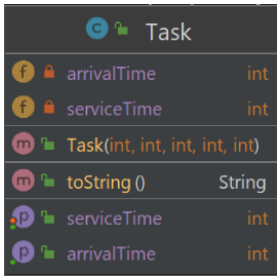
Diagrama UML generată direct din IntelliJ:



4. Implementarea

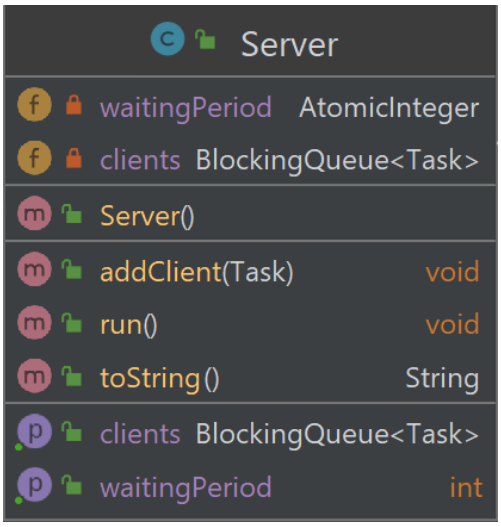
Pentu realizarea acestui proiect am folosit 9 clase:

- Clasa “Task”



Clasa “Task” reprezintă clasa pentru client care este caracterizat de timpul de sosire în coadă și timpul de servire, adică timpul cât așteaptă până să părăsească coada. Aceste 2 timpuri sunt setate random în constructor: timpul de sosire este un număr random ce se află între minArrivingTime și maxArrivingTime, iar timpul de servire este și el un număr random între minProcessingTime și maxProcessingTime.

- Clasa “Server”



Clasa “Server” se ocupă de gestionarea unei cozi ce conține mai multe task-uri (clienți).

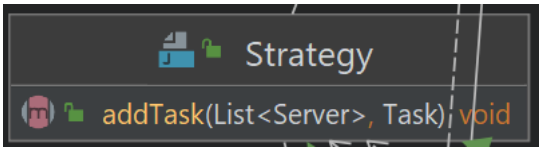
Astfel, un server(coadă) este caracterizată de un BlockingQueue de clienți și de un waitingPeriod ce reprezintă timpul în care coada este activă(are clienți în ea), adică suma timpurilor de servire a tuturor clienților din coadă la momentul respectiv.

Clasa mai conține și o variabila statică “averageWaitingPeriod” ce însumează timpul de așteptare al fiecui client ce intră în coadă. Aceasta va fi utilă pentru calcularea timpului mediu de așteptare.

Clasa implementează interfața Runnable, deci practic fiecare coadă este un thread, tocmai pentru a putea funcționa în paralel.

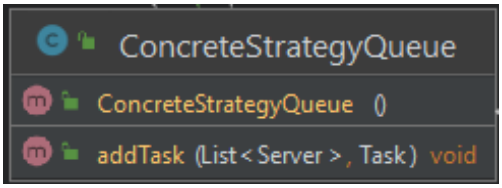
În metoda suprascrisă run() se simulează servirea clientului ce se află primul în coadă. Astfel, se ia clientul care e primul în coadă și odată cu trecerea timpului(1secunda) timpul de servire scade, adică scade timpul cât acesta stă în coadă și implicit scade și timpul în care coada este activă.

- Interfața “Strategy”



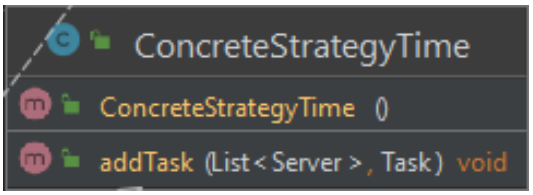
Aplicația se bazează și pe o interfață pentru strategie care va decide după ce criteriu clienții vor fi adăugați în coadă.

- Clasa “ConcreteStrategyQueue”



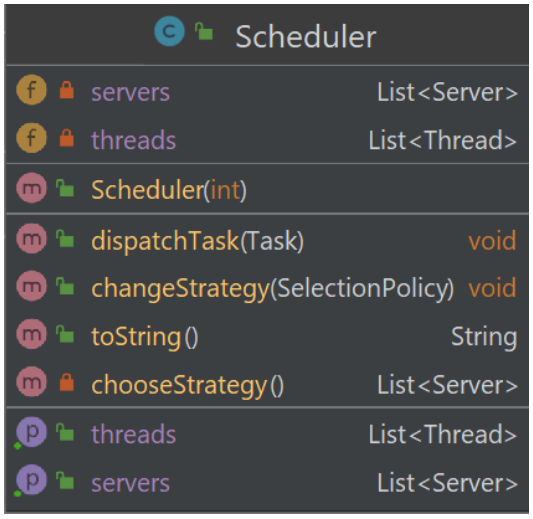
Această clasă implementează interfața “Strategy” de mai sus și constă în adăugarea unui client în cea mai scurtă coadă. Metoda suprascrisă caută în lista de servere această cea mai scurtă coadă și acolo introduce task-ul.

- Clasa “ConcreteStrategyQueue”



Această clasă implementează interfața “Strategy” de mai sus și constă în adăugarea unui client în coada care are timpul de servire cel mai scurt. Metoda suprascrisă caută într-o listă de servere această coadă cu timpul de procesare minim și acolo introduce task-ul.

- Clasa “Scheduler”



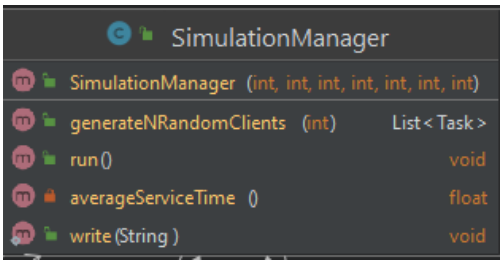
Clasa “Scheduler” se ocupă de gestionarea cozilor și a clienților care intră în ele. Așadar, în constructor se generează cozile(serverele) și creează thread-urile corespunzătoare și le pornește. Tot în constructor se alege strategia ce va fi respectată pentru a adăuga clienții în cozi.

Metoda “changeStrategy” schimbă strategia ce va fi aplicată, în funcție de starea cozilor.

Metoda “chooseStrategy” decide care strategie va fi folosită pentru a adăuga un client. Așadar, aplicația alege să adauge un client în coada cea mai scurtă, iar dacă există mai multe astfel de cozi, atunci se va schimba strategia și se va adăuga clientul în una dintre aceste cozi care are timpul de procesare cel mai scurt.

Deciderea cărei strategii va fi aplicată și adăugarea clientului se face propriu-zis în metoda dispatchTask() care se folosește atât de „chooseStrategy” cât și de „changeStrategy”.

- Clasa “SimulationManager”



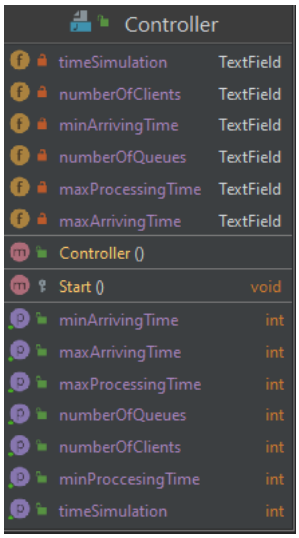
Clasa “SimulationManager” se ocupă de simularea propriu-zisa a sistemului de cozi. În constructor se seteaza parametrii de simulare: numărul de cozi, numărul de clienți, timpul minim de sosire, timpul maxim de sosire, timpul minim de procesare, timpul maxim de procesare și timpul în care simularea rulează. Tot în constructor golesc fișierul în care voi pune rezultatele simulării prin ștergerea lui dacă acesta există și crearea lui.

Constructorul se ocupă și de generarea random a clienților prin apelarea metodei “generateNRandomClients(). În această metodă, după ce se generează cei N clienți aceștia sunt sortați în ordinea crescătoare a timpului de sosire, pentru a facilita intrarea în cozi.

Clasa implementează interfața Runnable pentru că simularea se va face pe un thread. În metoda suprascrisă run() se face adăugarea clienților în cozi în funcție de timpul de sosire al fiecăruia și respectând strategia corespunzătoare. Tot aici calculez și afișez timpul mediu de așteptare cu ajutorul variabilei statice din clasa “Server”, dar și timpul mediu de procesare cu ajutorul metodei “averageServiceTime”. În această metodă calculez ora de vârf, adică intervalul de timp când se află cei mai mulți clienți în cozi. Dacă există mai multe astfel de intervale, îl iau pe primul. Pentru calculul capătului minim al intervalului am ales primul timp în care se atinge numărul maxim de clienți, iar pentru calculul capătului maxim al intervalului am adăugat la capătul minim timpul de servire al primului client care va ieși din oricare coadă.

Metoda “write()” scrie rezultatele simulării atât în consolă cât și în fișierul a cărui cale este speficicată. Scriere se realizează în timp real în metoda „run()” a acestei clase.

- Clasa “Controller”



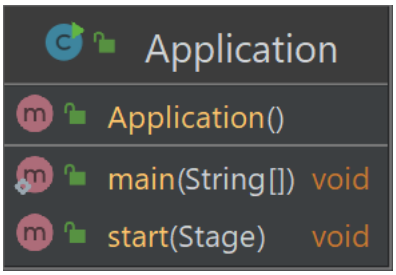
Clasa “Controller” preia practic comenzile utilizatorului asupra câmpurilor text și asupra butonului de “Start” din interfața grafică și le execută.

Câmpurile text reprezintă parametrii simulării: numărul de cozi, numărul de clienți, timpul minim de sosire, timpul maxim de sosire, timpul minim de procesare, timpul maxim de procesare și timpul în care simularea rulează.

Cu ajutorul ultimelor 7 metode din imagine se returnează sub formă de numere întregi valorile ce au fost introduse de utilizator in câmpurile text.

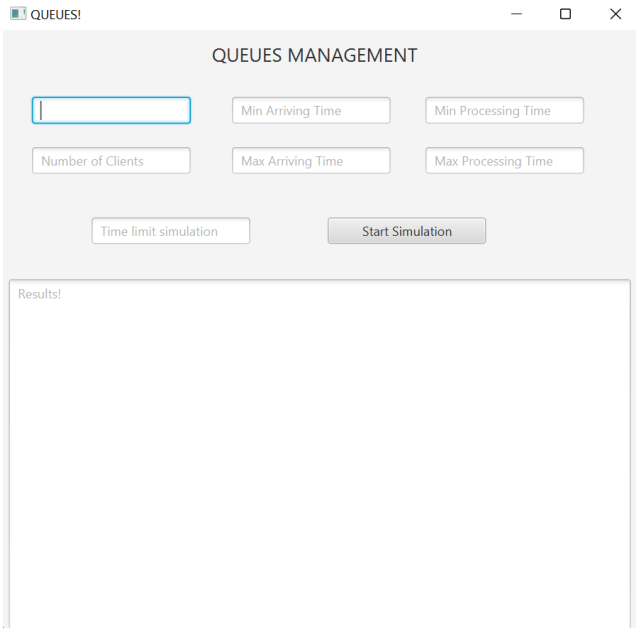
Prin apăsarea butonului de Start simularea pornește și pornește și un thread care se ocupă de afișarea în timp real a rezultatelor. Acest lucru se realizează în metoda “run()” ce se află în metoda “start()” care pornește simularea. Când simularea pornește se crează un obiect de tip “SimulationManager” al cărui constructor primește ca parametrii valorile de tip întreg returnate de cele 7 metode de care am amintit mai sus.

- Clasa “Application”



Clasa “Application” conține metoda “start” care creează fereastra aplicației, setându-i un titlu și adaugă prin intermediul unui fisier “.fxml” toate câmpurile și butonanele descrise în clasa “Controller”, această clasă fiind astfel legată și controla de clasa “Controller”.
Prin metoda “main” se execută aplicația.

- Implementarea interfeței utilizator



Interfața utilizator am creat-o utilizând JavaFx, întrucât oferă o gamă largă de opțiuni de personalizare a elementelor cum ar fi butoane, TextFields, TextAreas, etc. Mai mult decât atât este foarte ușor de lucrat cu JavaFx întrucât se bazează pe Drag and Drop și oferă și o previzualizare a produsului final.
Atât clasa “Controller” cât și clasa “Application” importa biblioteci din javafx, astfel că practic interfața este dată de conectarea și comunicarea celor doua clase.

5. Rezultate

Pentru testare am folosit 3 seturi de date pentru parametrii de simulare și am stocat rezultatele in 3 fișiere text "test1", "test2", "test3" stocate în folderul "resources"

Test 1	Test 2	Test 3
N = 4 Q = 2 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 30] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [2, 4]	N = 50 Q = 5 $t_{simulation}^{MAX}$ = 60 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [2, 40] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [1, 7]	N = 1000 Q = 20 $t_{simulation}^{MAX}$ = 200 seconds $[t_{arrival}^{MIN}, t_{arrival}^{MAX}]$ = [10, 100] $[t_{service}^{MIN}, t_{service}^{MAX}]$= [3, 9]

6. Concluzii

Realizând această primă temă mi-am sedimentat mai bine cunoștințele legate de conceptele programării orientate pe obiect ceea ce îmi va fi foarte de folos pentru următoarele proiecte. De asemenea, am învățat să lucrez cu Thread-uri și am văzut importanța lor, eficientizând procesul de gestiune a cozilor prin rularea lor în paralel.

Prin intermediul acestei teme am descoperit cât de important este ca în unele cazuri să se lucreze cu thread-uri, deoarece se eficientizează soluția astfel prin execuția mai multor programe în paralel.

Posibile dezvoltări ulterioare

- găsirea unei noi strategii/implementări care să eficientizeze și mai mult acest proces de așezare în coadă și servire
- îmbunătățiri la nivel de interfață grafică.

7. Bibliografie

1. Stack Overflow

2. YouTube

3. JUnit 5 | IntelliJ IDEA (jetbrains.com)